Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

A general-purpose modifiable AST parser

Michael Doberstein

Bachelorarbeit

Beginn der Arbeit:	29th of September 2024
Abgabe der Arbeit:	30th of December 2024
Gutachter:	Dr. John Witulski
	Dr. Jens Bendisposto

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30th of December 2024

Michael Doberstein

Abstract

Fassen Sie die Fragestellung, Motivation und Ergebnisse Ihrer Arbeit hier in wenigen Worten zusammen.

Die Zusammenfassung sollte den Umfang einer Seite nicht überschreiten.

Acknowledgements

Im Falle, dass Sie Ihrer Arbeit eine Danksagung für Ihre Unterstützer (Familie, Freunde, Betreuer) hinzufügen möchten, können Sie diese hier platzieren.

Dieser Part ist optional und kann im Quelltext auskommentiert werden.

Contents

1 Introduction 1

1.1 Motivation 1

1.2 Related work 1

1.3 Tools used 1

2 Prerequisites 2

2.1 Regular Expressions and Grammars 2

2.2 Lexer 3

2.3 Parser 3

2.3.1 LR(1)-Parsing 4

2.3.2 Preprocessor Statements 5

3 Implementation 5

3.1 Class generation vs general classes 5

3.2 Extendability 5

3.3 Grammar file syntax 6

3.4 CST-Generation 6

3.5 AST-Generation 7

3.5.1 list-Modifier 8

3.5.2 alias-Modifier 9

3.5.3 hidden-Modifier 10

3.5.4 inline-Modifier 10

3.5.5 Not implemented modifiers 10

3.6 AST search methods 10

3.7 Selectors 11

3.8 AST Modifications 12

4 Proof of concept **13**

4.1 Java Subset 13

4.2 Automated Refactoring 13

5 Conclusion **13**

List of Figures **14**

List of Tables **14**

1 Introduction

1.1 Motivation

Software developers are often facing the issue of maintaining large amount of source code. The keep the source code up to date, often times a process called refactoring is needed, in which the software developers update the code.

To allow the software developers to perform these updates automatically, most integrated development environments (IDE) provide tools to perform predefined refactorings automatically. The more source code a developer has to maintain, it gets increasingly difficult to perform a refactoring. One way to create automated custom refactorings is by parsing the source code and building it into an abstract syntax tree (AST), which can be used to manipulate the source code.

The tools given by an IDE or other sources are sometimes not sufficient to perform an automated refactoring, therefore other methods of generating custom refactorings are required.

Here we provide a tool, which can parse any programming language based on their corresponding grammar definition and generates a AST which can be modified by the user and afterwards reverted back to the source code.

TODO: Discussion

1.2 Related work

There are many parser generators that have a similar approach, but different goals. One major example is ANTLR, which generates a parser for a given grammar. The generated parsers are mainly used for further use in a compiler. This usecase differs from the usecase of refactoring, as the compiler does not need whitespaces, comments or preprocessor statements in most cases. In the use case of refactoring, we need to preserve all of these tokens to reconstruct the source code as accurately as possible. We do not want to loose all comments due to a refactoring beeing made. Also, this thesis provides a tool which provides an API specifically aimed to refactor source code.

1.3 Tools used

The machine learning assistant of the IntelliJ IDE was used while writing the source code. DeepL Write was used to check the grammar and spelling of the thesis.

2 Prerequisites

In this chapter we want to establish the prerequisites required to understanding this thesis.

2.1 Regular Expressions and Grammars

In theoretical computer science, a grammar consists out of a set Σ of terminal symbols, a set N of nonterminal symbols, a set P of productions and a start symbol S. S is always a nonterminal symbol and therefore $S \in N$. The productions are defined as $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$. The $^+$ refers to one or more, the * refers to zero or more.

Grammars can be seperated in different different classes based on the chromsky hierarchy. The main ones we are looking at in this thesis are the contextfree grammars and the regular grammars.

A grammar is context free, if for all productions $p \rightarrow q$ in P the following applies: $p \in N$.

A grammar is a regular grammar, if for all produtions $p \rightarrow q$ in p the following applies: $p \in N$ and $q \in \Sigma \cup \Sigma N$.

Grammars are used, because they can describe a language. In our use case, they describe an programming language and can be used, to parse the source code. Regular expressions are short statements, that avoid defining an entire grammar definition. Their downside is, that they are not able to describe alot of the features modern programming languages have. For example, it is not possible to write a regular grammar for the language that has the same amount of opening and closing brackets. An example for a regular expression would be this:

$$a^*ba^*b$$

This would parse all words that have any amount of "a", followed by exactly one b, followed by any amount of "a", followed by exactly one b.

To handle these situations, we require the context free grammars, they are able to parse more cases and are sufficient enough, to parse most programming languages. However, even the context free grammars are not powerfull enough to correctly verify if a given source code is valid. For this, we would technically need context sensitive grammars. Those come with the cost of a worse runtime, therefore usually context free grammars are used when writing compilers and additional verifications are done after the parsing to ensure the language is indeed valid source code.

An example for a context free grammar:

Let $\Sigma = \{E\}, N = \{a, b\}, S = \{E\}$ and the productions defined as:

$$\begin{aligned} P = \{E &\rightarrow aEb, \\ E &\rightarrow \epsilon\} \end{aligned}$$

ϵ is a special symbol, it refers to the empty word. That means, that E can be derived to nothing. Without this, we would have endless self recursion in the grammar rule.

This grammar would be equivalent to the opening and closing bracket example from previously. This will parse any string, which has for one opening a exactly one opening b.

2.2 Lexer

The parse process of a compiler is often times split into two parts. The first part is the so called lexer. The task of the lexer is to take in the entire source code and create a so called token stream. The lexer usually can mostly run with regular expressions and avoids context free grammars for the most part.

The lexer is a preparation for the parser, as it removes unwanted characters like whitespaces and generalizes other tokens by already classifying them for their intended use case. For example could be any string matching the regular expression `[a-zA-Z_][a-zA-Z_0-9]*` an identifier and `[0-9]+` would be an integer. The name identifier is often used for variable, function and classnames in lexers. By doing this, the parser later only needs to work with the abstract identifier, not with the real input.

2.3 Parser

The second part will perform the parsing on the token stream and by using the grammar rules. There are many kinds of parsers, the most common ones used in compiler development are:

1. LL(1): Scanning the input from left to right, applying leftmost deriviation, using one token of lookahead.
2. LR(1): Scanning the input from left to right, applying rightmost deriviation, using one token of lookahead.
3. LALR(1): A variation of LR(1), which reduces the amount of calculations and memory requirements.

There are two goals of the parsing process. The first one is to verify that the input matches the grammar. The second one is to generate a tree structure, which represents the input. The tree structure generated by the parser is called a concrete syntax tree (CST). This tree structure is very verbose and represents the grammar definition exactly.

There are two main different ways used to parse source code. The first approach is top-down parsing. In top-down parsing, the root node will be generated first, following by the childrens and the leaf nodes will be generated at the end. LL(1) parsing is one example for top-down parsing. The other approach is bottom-down parsing. In this parsing strategy the leaf nodes are generated first and the root node is generated at the very end. LR(1) and LALR(1) are examples for bottom-down parsing algorithms.

The top-down parsers have the benefit of being rather easy to understand, however they are less powerfull. Bottom-down parsers are harder to understand and to debug, however they can parse more grammars.

Each of the presented grammars are not able to parse all context free grammars, but they are sufficient enough to parse most programming languages. LR(1) is the parsing strategy that is able to parse the most grammars, but requires the largest amount of memory and calculations to correctly parse.

As memory and runtime is not the main priority in this thesis, a LR(1) parser was implemented to allow the largest amount of grammars being parsed.

2.3.1 LR(1)-Parsing

The LR(1) parsing strategy relies on two different tables. The first table is called an action table. The rows represents different states. The columns represent the terminal symbols of the grammar. The value of each cell can be empty, a shift, a reduce or an accept action.

A shift action will consume a token from the token stream and move to a new state.

A reduce action will go back to a previously encountered state and references a grammar rule that is beeing reduced.

A accept action is only defined once in the entire table. Once this is encountered, the parsing process is finished and the input was successfully parsed.

The second table is the goto table. The rows represents the different states. The columns represent the non terminal symbols of the grammar. The content of each cell can either be a reference another state or empty.

The parser itself manages a stack of states. The top of the stack is the current state that is currently being processed. The parser reads the current token in the token stream provided by the lexer and receives the current action from the action table. Based on the action type, different behaviours apply.

On a shift action, the parser will shift the position in the token stream and will continue in the next iteration with a new token. The new state defined in the shift action will be pushed on the stack. Then the next iteration is started.

On a reduce action, the parser will pop as n states from the stack, where n is equal to the amount of terminal and nonterminal symbols on the right hand side of the grammar rule. Afterwards, the new top of stack is beeing read and the state defined in the goto table of the top of stack and the current token in the token stream will be pushed on the stack.

On an accept action, the parser will finalize the parsing process accept the input.

2.3.2 Preprocessor Statements

Hier kurz darauf eingehen, was preprocessor statements sind.

3 Implementation

3.1 Class generation vs general classes

Mostly, parser generators will create classes for every production. This allows the user to modify the generated classes, for example adding new methods. This allows for better usablility if done correctly. We chose to not follow this approach, instead we have a small set of predefined classes, which are used across all nodes. With this approach, no additional code generation for the parser is needed. The parser can run standalone and in place. This approach allows for an easier introduction to the system, as a grammar and a source file can simply be passed to the parser generator and an AST will be generated. Otherwise, first the parser needs to be generated and afterwards the parser needs to be called to parse the source files.

3.2 Extendability

Hier darauf eingehen, in wie weit das Programm erweiterbar ist. Welche Möglichkeiten existieren in den Lexer/Parser einzugreifen?

3.3 Grammar file syntax

The productions and lexer definitions are defined seperately from the implementations in their own text files. Each language has exactly one text file defining the productions and lexer definitions. The contents are seperated in four different chapters.

The first chapter is `LANGUAGE_DEF`. This contains various general definitions for the language. This chapter contains e.g. what comment styles are allowed and how they are identified, name of the language, file extension used, wether or not the language is case sensitive and what production should be used as the starting production. All settings here are defined as key, value pairs.

The second chapter is the `LEXER_RULES`. In there, we define all tokens that can appear in the source file and that the lexer should handle. The lexer can handle three different types of definitions. The first type is matching a fixed string. The second type matches a regex. The last type uses a custom implementation to check for a match. The last matching matching definition wins. For example, the token definition `true_literal = "true";` and the definition `identifier = regex([a-zA-Z_][a-zA-Z_0-9]*);` would both match the string `true`. Therefore, the more general `identifier` token should be defined before the `true_literal` token. This will cause the lexer to identify the text `true` as `true_literal`. A lexer key needs to be written in all lowercase (and lowercase) letters.

The chapter `HIDDEN_LEXER_RULES` is a comma seperated list of lexer definitions keys, which should not be handled in the parse process and also not be visible in the AST by default.

The chapter `GRAMMAR_RULES` contains all productions of this language. The syntax is similar, but not equal, to the backus natus form. (TODO: Source). The syntax was extended by most of the modifiers proposed by [**?**]. The terminals are the keys of the lexer definitions. The nonterminals are defined by the left hand sides of the productions. All nonterminals have to be written in uppercase and underscores. `EPSILON` is used for the empty word. Every terminal and nonterminal symbol can be extended by a list of modifiers in square brackets after the name. `FORMAL_PARAMETER -> TYPE identifier[alias=parameterName];` defines the production with `FORMAL_PARAMTER` on the left hand side and `TYPE identifier` on the right hand side. `TYPE` references a different production. `identifier` references a lexer definition. `[alias=paramterName]` sets the alias modifier onto the `identifier` symbol.

3.4 CST-Generation

The concrete syntax tree (CST) is a tree structure, that matches the the grammar and lexer definitions exactly. Our CST also should contain nodes, which are usually omitted, like comments or whitespaces.

The CST is implemented by a `ConcreteSyntaxTreeNode` class. A node can either reference a production or a lexer token. We maintain the production, as these allow us to perform advanced modifications and generate a less verbose abstract syntax tree from the cst. The lexer tokens are maintaned, as these include the original sources.

This structure is generated by the parser directly. The parser manages a second stack, containing tree nodes that are not attached to a parent node.

As the parser is progressing through the token stream provided by the lexer, it generates new tree nodes and pushes them on the stack whenever either the token is contained in `HIDDEN_LEXER_RULES` or the action performed is a shift action.

On a reduce action, the parser will also create a new tree node. This tree nodes will have other tree nodes from the top of the stack as children. This behaviour is very similar to the behaviour how the state stack of the parser manages its state. The main difference are the tokens in the `HIDDEN_LEXER_RULES`. The reduced production might only have one symbol on the right hand side, but on the tree node stack are two nodes from `HIDDEN_LEXER_RULES`. In this case, the parser will pop three states from the top of the tree node stack and add them as children of the newly created node in reverse order. In case the parser reaches the `acc` state, it will not be guaranteed that the stack only contains the root node though. There might be tokens from `HIDDEN_LEXER_RULES`, which appeared at the start of the parsed string. Therefore, the parser will find the root node, and prepend all remaining entries in the tree node stack to the childrens of the root node.

With this algorithm, every single token from the token list of the lexer will be transformed into a tree node and will be in the CST. This guarantees, that the entire source code is represented in the CST and the CST could be transformed back to the source code exactly.

3.5 AST-Generation

The AST consists of three different node types. `ProductionTreeNode` are nodes that originated from a grammar production. `TokenTreeNode` are nodes that originated from a lexer token. `StringTreeNode` are nodes that contain a string and are used to replace other tree ndoes. These nodes are usually not part of the initial AST, they can be added later on to the AST.

The AST is created from the CST. The `TokenTreeNode` which reference a token from `HIDDEN_LEXER_RULES` will marked as hidden. These nodes are stored in the AST, but they will not be visible to the user, unless specifically requested. Afterwards, the AST structure will be modified by the modifiers defined in the grammar file. Each modifier has their own implementation and will be applied bottom up for each node.

3.5.1 list-Modifier

The `list` modifier, purposed by [**?**], is a modifier that is supposed to flatten out self recursive productions. This modifier can only be applied to symbols on the left hand side of a production. The following example might be a production for the function paramters in common programming languages.

```
FORMAL_PARAMETER_LIST[list] -> FORMAL_PARAMETER_LIST FORMAL_PARAMETER | FORMAL_PARAMETER | EP
```

Without the `[list]` modifier, this would create a tree structure like this

1. `FORMAL_PARAMETER_LIST`
    1. `FORMAL_PARAMETER_LIST`
        1. `FORMAL_PARAMETER`
    2. `FORMAL_PARAMETER`
2. `FORMAL_PARAMETER`

Mostly, we do not want a structure like this. By applying the `list` modifier, the resulting structure will be:

1. `FORMAL_PARAMETER_LIST`
    1. `FORMAL_PARAMETER`
    2. `FORMAL_PARAMETER`
    3. `FORMAL_PARAMETER`

This structure is much easier to manage and still contains all relevant informations about the sources.

The modifier is applied by checking if the children of the current `ProductionTreeNode` n contains a `ProductionTreeNode` m referencing the same production. If this is the case, m is replaced in the children list of n by all childrens of m.

3.5.2 alias-Modifier

The `alias` modifier, purposed by [**?**], is a modifier that can be applied to any symbol in the right hand side of a production. This modifier, will add an alias to the tree node, which can be used to search the tree. The following might be an example for a production for addition in a programming language.

```
ADD -> NUMBER[alias=left] plus NUMBER[alias=right];
```

Without the alias modifier, the only way to differentiate between the both `NUMBER` nodes would be by the order of the children of the `ADD` tree node. The `alias` modifier allows for cleaner searches in the AST.

This modifier is applied by simply storing the alias in the tree node.

3.5.3 hidden-Modifier

The `hidden` modifier, purposed by [**?**], is a modifier that can be applied to terminal symbols in the right hand side of a production. This modifier will hide the tree node in the AST. In the previous example of the `ADD` production, the `ADD` tree node would have a `TokenTreeNode` child, which references the `plus` lexer definition. This information is obsolute, as the production will always contain this node and the production name already contains the necessary informations. By applying the `hidden` modifier to the `plus` symbol, the corresponding tree node will still be present, but not visible unless specifically requested.

3.5.4 inline-Modifier

The `inline` modifier, purposed by [**?**], is a modifier that can be applied to nonterminal symbols in the right hand side of a production.

By applying the modifier, the node will be replaced with all its children. This modifier should be used on nonterminals, which itself do not carry important informations, but their children do. This way all informations are maintaned, but the tree structure gets simplyfied.

3.5.5 Not implemented modifiers

[**?**] purposed additional modifiers, those are not implemented currently. That would be the `Boolean Access` modifier, which would replace a node with a boolean value. However, as we do not generate classes for each production, this would serve little purpose for us. The same reason applies to the `superclass` modifier, which would create a hierarchy in the generated classes of the parser generator.

3.6 AST search methods

Before any modification can be applied to the AST, firstly the nodes have to be found which can be modified.

For this approach, each ast node has three methods.

1. `query: Selector -> QueryResult`
2. `queryChildren: Selector -> QueryResult`
3. `queryImmediateChildren: Selector -> QueryResult`

The `query` methods takes a selector and returns a `QueryResult`. The `QueryResult` contains all nodes in the subtree of the searched node which match the `selector`. The `QueryResult` can also contain the searched node itself.

The `queryChildren` behaves the same, but will not include the searched node itself.

The `queryImmediateChildren` will only include the immediate children of the searched node matching the `selector`.

The `QueryResult` instance also allows to perform queries on the result. This will perform the according method on all nodes in the result and create a new `QueryResult` instance containing the merged results of each node. This allows for easy chaining of selectors. Also `QueryResult` instances can be merged for further processing with the `merge` function.

3.7 Selectors

A `Selector` is a classinstance, which should test wether a given treenode matches the rules defined by the `Selector`. The `Selectors` can be splitted into three different categories.

The first category are logical `Selectors`. They should represent logical operations. There are two `Selectors` defined by default, `AndSelector` and `OrSelector`. Each of them takes a list of other selectors. The `AndSelector` will only match tree nodes, that match all passed `Selectors`. The `OrSelector` will match all tree nodes, where atleast one of the passed `Selectors` match the node.

The second category are data `Selectors`. Those selectors search for nodes based on the data contained in them. These can match the name of a production, the alias of a production, the name of a lexer definition and the value of a lexer token. The implemented selectors are `AliasSelector`,`ProductionSelector`,`TokenSelector`,`TokenValueSelector`. All of those receive a string as a parameter and check the corresponding notes for this string.

The last category are structural `Selectors`. Those selectors search for nodes that have specific nodes in the children or parents.

The following example of an AST should show the purpose of this category.

1. `METHOD_CALL`
    (a) `CALLED_ON`
        i. `translator`
    (b) `METHOD_NAME`
        i. `translate`

To search for all `METHOD_CALL` nodes, that call translate on a translator object, we will want these types of selectors.

An example selector for this would be

```
new AndSelector(
new ProductionSelector("METHOD_CALL"),
new HasImmediateChildSelector(
new AndSelector(
new ProductionSelector("CALLED_ON"),
new HasImmediateChildSelector(
new TokenValueSelector("translator")
)
)
),
new HasImmediateChildSelector(
new AndSelector(
new ProductionSelector("METHOD_NAME"),
new HasImmediateChildSelector(
new TokenValueSelector("translate")
)
)
)
)
```

By calling the `query` method on the root node of the AST, we will receive all `METHOD_CALL` nodes in the AST matching these conditions.

3.8 AST Modifications

The `AbstractSyntaxTreeNode` class contains multiple methods to make modifications.

1. `replace` Replaces this node with a list of other nodes.
2. `replaceChild` Replaces a node in the childrens with a list of other nodes.
3. `remove` Removes this node
4. `removeChild` Removes a child
5. `addChild` Adds a child to the end of the children list
6. `addChildBefore` Adds a child before another node.

7. `addChildAfter` Adds a child after another node.
8. `deepClone` Clones this node and all childrens, returns a node that equals the current one but without a parent reference.

Each `AbstractSyntaxTreeNode` references their corresponding parent node. To maintain integrity, you can only add nodes that have no parent node and replace only nodes with a parent reference. To add a node to multiple other nodes, you can clone the node to be added.

3.9 Extendability

To allow for the largest amount of possible extendability, we allow for various ways of customization

3.9.1 Lexer

As mentioned before, the lexer definitions can contain custom matchers. Those matchers are custom implementations, which are referenced by a name in the grammar file. These classes are written by the user and registered by calling `CustomMatcherRegistry.registerCustomMatch` A custom matcher needs to implement the `ICustomMatcher` interface. A matcher gets a `LexerContext` and returns `null` or the matched string. By the `LexerContext` parameter, the matcher can receive the remaining input and checks if the beginning of the remaining input can be tokenized by this matcher.

3.9.2 Parser

???

3.9.3 AST Generation

We allow for various ways for custimizing the AST generation. Firstly, custom modifiers for grammar symbols can be implemented. Also, the AST can be postprocessed and decorated. For example, the tree ndoes can be replaced with their own implementation, they simply need to extend `AbstractSyntaxTreeNode`.

3.9.4 Selectors

Custom selectors can be defined by implementing `ISelector`. Those can simply be defined and used with the query methods of the tree node.

4 Proof of concept

4.1 Java Subset

Hier kurz darauf eingehen, für welche Sprache eine Grammatik definiert wurde.

4.2 Automated Refactoring

Hier Beispiel/Beispiele für automatisierte Refactorings anführen. Beispielsweise Funktionsnamen von underscore zu camelCase überführen.

5 Conclusion

Am Ende der Arbeit werden noch einmal die erreichten Ergebnisse zusammengefasst und diskutiert.

List of Figures

List of Tables